NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AUTOMATED CARTOGRAPHY BY AN AUTONOMOUS MOBILE ROBOT

by

Mark Merrell

March 1999

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
Automated Cartography by an Autonomous Mobile Robot

5. FUNDING NUMBERS

6. AUTHOR(S)
Mark L. Merrell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The major goal of this thesis was to create a map of a room by an autonomous mobile robot using the robot's internal odometry measurements and ultrasonic sensors. Yamabico, an autonomous mobile robot, will be controlled by Model-based Mobile robot Language (MML).

The research for this thesis included the development of an algorithm to use information from the line-fitting capability of MML. It also included research about the inherent errors that are incurred using sonar for precise measurements.

The results of this thesis are that it is possible for a map to be created by an autonomous robot using only ultrasonic sensors. However, the results would be much more accurate if an external source of navigation was used to correct the errors inherent in ultrasonic sensors and the robot's odometry.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Automated Cartography, Autonomous Mobile Robots, Ultrasonic Sensors	74
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AUTOMATED CARTOGRAPHY BY AN AUTONOMOUS MOBILE ROBOT

Mark L. Merrell
Captain, United States Army
B.S., United States Military Academy, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Author: _____
Mark L. Merrell

Approved by: _____
Yutaka Kanayama, Thesis Advisor

_____
Thomas Wu, Second Reader

_____
Dan Boger, Department Chairman

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The major goal of this thesis was to create a map of a room by an autonomous mobile robot using the robot's internal odometry measurements and ultrasonic sensors. Yamabico, an autonomous mobile robot, will be controlled by Model-based Mobile robot Language (MML).

The research for this thesis included the development of an algorithm to use information from the line-fitting capability of MML. It also included research about the inherent errors that are incurred using sonar for precise measurements.

The results of this thesis are that it is possible for a map to be created by an autonomous robot using only ultrasonic sensors. However, the results would be much more accurate if an external source of navigation was used to correct the errors inherent in ultrasonic sensors and the robot's odometry.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

Most humans have a natural ability to navigate in an unfamiliar environment. Robots do not. Robots require a tremendous amount of logic to control its motion, activate sensors, analyze feedback from the sensors, determine the best path and navigate in the environment.

The need to navigate in its environment is one task that is common to all autonomous mobile robots. To navigate from one location to another, the robot must have a representation of its environment stored in its memory. People use a map to represent the world around him. One common representation for robots is the use of a grid system that designates each grid as being occupied or not. Another representation is to designate each wall as a line segment that is defined by two end points. The environment can be defined by a list of points. The robot control language developed by Professor Yutaka Kanayama uses line segments. This language is called Model-based Mobile robot Language (MML). MML will be described in more detail in Chapter III.

A robot must have a way to sense its surroundings. It must be able to determine where it is in the environment so that it may go to other places in that environment. There are many kinds of sensors. Some are internal to the robot, such as sonar, infrared, internal odometry, and laser range finders. Some make use of a known reference point that is external to the robot. These include a magnetic compass, special tape on the floor that the robot can read, radio waves that transmit a signal that the robot can listen to and triangulate its position, and the global positioning system (GPS). Sensors can be used for

map making or for obstacle avoidance when moving around in the environment. External sources are usually more accurate because the robot knows exactly where the external source is so it can make a correction based on known points instead of its own internal sensors.

With a representation of the environment and sensors, the robot can determine where it is, navigate to a new location and update/correct its internal odometry on the way.

B. OVERVIEW

This thesis will use the robot Yamabico-11 and the robot control software Model-based Mobile robot Language (MML) to navigate in an unknown environment using ultrasonic sensors to determine the obstacles in a room and then download the results to the user.

Yamabico is the Naval Postgraduate School's autonomous differential-drive robot with shaft encoders for internal odometry. It has two independent drive wheel systems that allow it to rotate in place, or move forward or backward while making a turn with any degree of sharpness. Yamabico also has a CCD camera and 12 ultrasonic transducer pairs for navigation and obstacle avoidance.

The 12 sonar pairs are positioned 30 degrees apart starting with 0 degrees positioned at the front of the primary direction of travel. In each sonar pair, one is designated to be the sender, one is the receiver. The maximum range for the sonar is 409 centimeters. The minimum range is 10 centimeters. Sonars have many advantages and disadvantages, which will be discussed in Chapter IV.

The robot's motion is controlled by MML. This language is made up of many powerful functions written in C programming language. MML can define specific lines and curves to track and the language determines the optimum turning point to transition from one line to another. MML will be discussed in Chapter III.

This thesis will explain how an algorithm was developed and programmed into Yamabico to successfully navigate in an unknown environment and return the dimensions of the room to the user.

C. ORGANIZATION

Chapter II discusses the algorithms that were developed to complete this thesis. An algorithm was developed to use the information derived from the Generalized Least Fitting Squares algorithm.

Chapter III explains MML in more detail. Many advantages of MML are discussed. An explanation of the code written for this thesis will be discussed.

There are many hardware limitations that can cause errors and limit the accuracy of the map. Chapter IV discusses two items that cause the most errors in robot navigation. They are the sonars and errors caused by the physical movement of the robot around the room.

In the final chapter, the results of the experimentation and conclusion are discussed.

Appendix A lists the code that was developed specifically for this thesis. The file user.c is the program that controls the robot and moves it about in the room. The file follow.c has some of the administrative functions for user.c file. It also manipulates the

line segments derived from the Least Squares Fitting algorithm into a map that is usable by the robot.

Appendix B shows the proof that the algorithm to adjust the robot heading will work using the Generalized Least Squares Fitting algorithm.

D. PROBLEM STATEMENTS

The major goal of this thesis was to create a map of a room by an autonomous mobile robot using the robot's internal odometry measurements and ultrasonic sensors. Yamabico, an autonomous mobile robot, will be controlled by Model-based Mobile robot Language (MML).

The research for this thesis included the development of an algorithm to use information from the line-fitting capability of MML. It also included research about the inherent errors that are incurred using sonar for precise measurements.

The results of this thesis are that it is possible for a map to be created by an autonomous robot using only ultrasonic sensors. However, the results would be much more accurate if an external source of navigation was used to correct the errors inherent in ultrasonic sensors and the robot's odometry.

II. METHODS USED FOR IMPLEMENTING THIS THESIS

A. DISTANCE MEASURING TASKS

In order for the robot to determine where it is and where the wall is, it must use some sort of distance measuring device. The most common distance measuring device is undoubtedly sonar. It is popular because it is inexpensive and accurate under certain conditions. The functionality of sonars will be discussed more thoroughly in Chapter IV. Another method for determining the range to an object is with a laser range finder. This is much more expensive, but also much more accurate.

Sonars send a signal out and wait for the signal to return. The speed of the signal through a particular medium is known. The distance is calculated by multiplying the speed of the signal by half the time it takes the signal to make the round trip. These are called "Time of Flight" sensors. The methods proposed in this paper will work for any type of range finder, but for readability of this paper, sonars will be referred to throughout this thesis.

A robot can maintain its positional awareness by using its internal odometry. If the robot is in an unknown environment, the only means of determining its position is through its internal odometry. By measuring the change in the rotation of each wheel, the robot can calculate its current position and orientation in the global coordinate system. Coupling its own position and orientation with the direction and distance from the sonar return, the robot can determine the position of the object that was seen by the sonar. By running a line fitting algorithm on the sonar points, it is possible to find a line to represent

the object in the real world. This thesis will use the Generalized Least Squares Fitting method for determining the best fitting line.

Least Squares Fitting is a method for determining the line that best fits a series of points. It can be referenced from the global origin by an angle and a distance. To determine the angle and the distance, the fitted line is extended to infinity in both directions. The angle, α, is formed by the x-axis and a line perpendicular to the fitted line that goes through the global origin. The distance, r, is the distance from the global origin to the closest point on the line.

In least squares fitting, the angle, α, will always be a value from -90 degrees up to but not including 90 degrees. Figure 1 shows the possible range of values for α and r in the four quadrants. In the first and third quadrants, α is positive. The distance r tells you which direction from the origin to find the line. If the distance is negative, that tells you

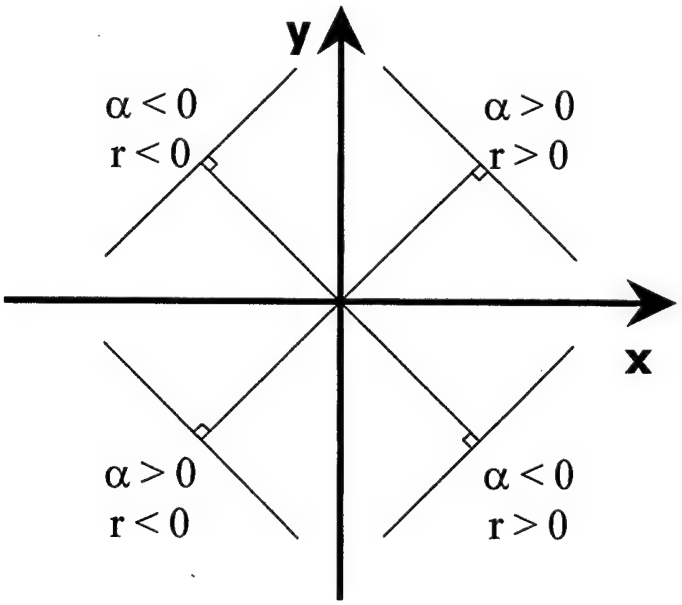

Figure 1. Example of angle and distance.

the fitted line is actually in the lower-left quadrant. The second and fourth quadrants also have similar properties.

B. ALGORITHM DESCRIPTION

Model-based Mobile robot Language (MML) is designed so that you tell the robot what line to track, and it will attempt to track it. By using sonar or internal positioning devices, if it notices that it is no longer on that line, it will make an adjustment to correct itself. The two figures in the left column of Figure 2 are possible examples of the robot tracking a line with an orientation of 0 degrees using sonar to help maintain its heading.

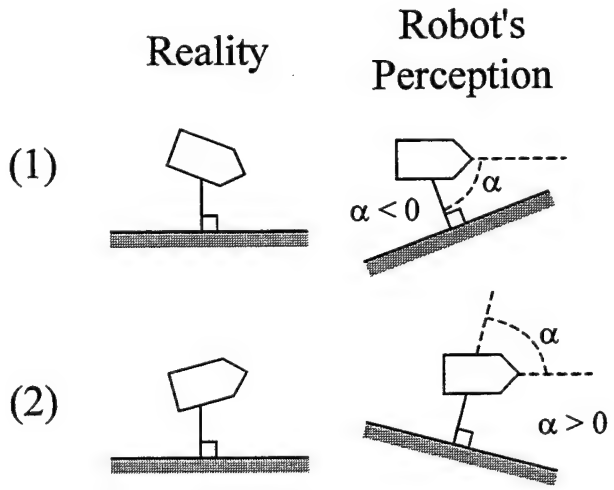

Figure 2. Robot's Perception of Reality.

An uneven surface or an imperfect internal odometry can introduce errors that could cause the robot to be heading a direction other than 0 degrees. The right two figures show how the robot believes it is still heading 0 degrees and the wall has shifted. It thinks it is still going straight and the wall has moved. Since we know in the real world that walls don't move, we can assume that position errors have been introduced. To correct this

problem, we tell the robot its actual heading. It calculates and executes the necessary correction to adjust the heading.

To understand more clearly how we can get α from the wall to the robot instead of the wall to the origin, refer to Figure 3.

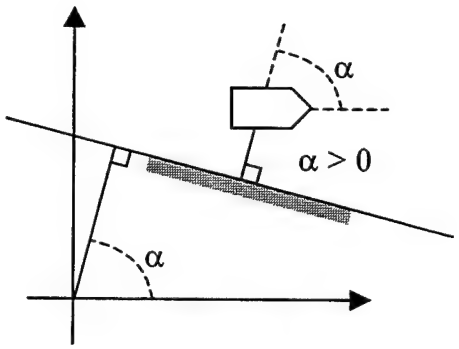

Figure 3. Calculation of alpha.

For clarity of this discussion, the walls will be labeled as wall 1 for the first wall to be tracked. For each left turn, the number of the wall will be increased by one. For each right turn the number of the wall will be decremented by one. In Figure 4, the wall at the bottom of the figure will be called wall 1. In a counter-clockwise fashion, the walls will be labeled wall 2, wall 3 and wall 4.

If the robot is tracking wall 1, which is 0 degrees, and it has a heading error of 10 degrees to the right, the α that is returned will be -80 degrees. If it is on track, α will be -90 degrees will be returned. A heading error of 10 degrees to the left will return 80 degrees as α. However, if you look at wall 2, you notice that if the heading error is 10 degrees to the right, α will return a value of 10. A correct heading while tracking wall 2 will return an α value of 0 degrees, and a heading error of 10 degrees to the left will give an α of -10 degrees. Wall 3 and wall 4 are similar to wall 1 and wall 2 respectively. By

looking at Figure 4, you will notice that even if the robot is tracking exactly parallel to the walls perfectly, the α that is returned will depend on which wall is being followed.

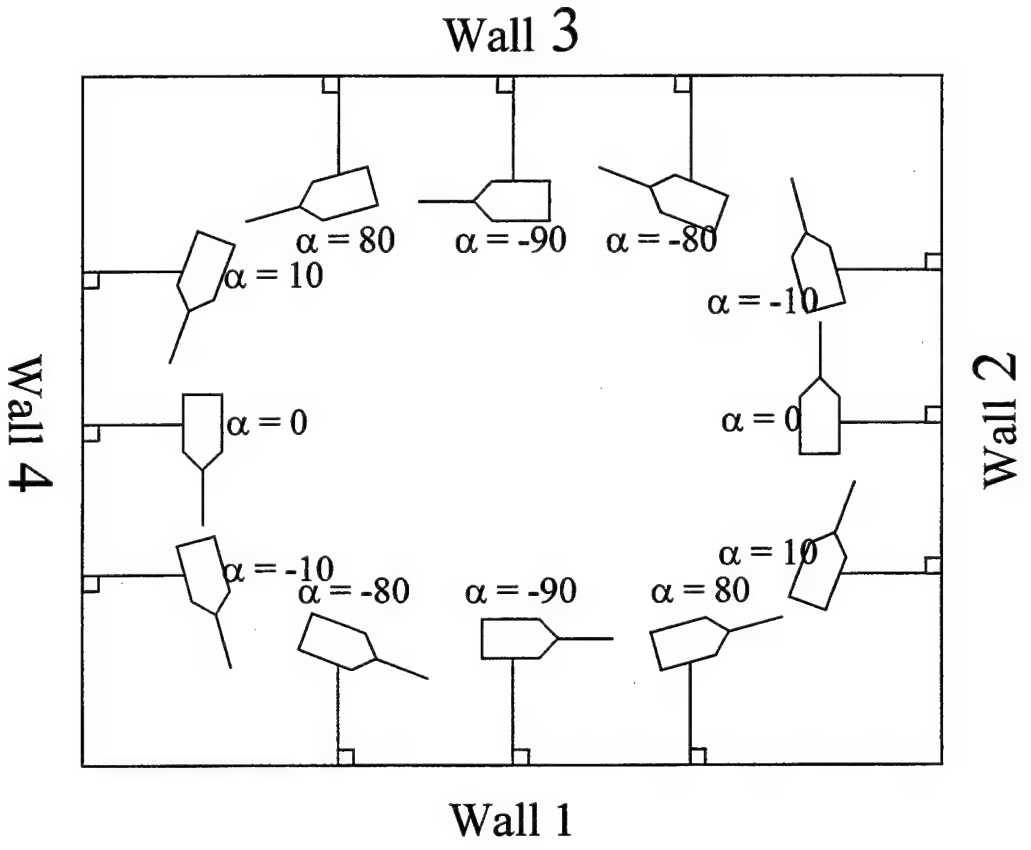

Figure 4. Example values of alpha near different walls.

The equation used to correct the heading will need to handle all cases of α. It is possible to correct the tracking error while tracking wall 2 and wall 4 by subtracting the returned α from the heading the robot is supposed to be tracking. On these two walls, if the robot is tracking the wall perfectly, α will be 0. A line of code to correct the heading would look like this:

```
Actual_Heading = Wall_Direction - alpha,                    (1)
```

where `Actual_Heading` is the heading the robot is actually following and `Wall_Direction` is the direction of the wall. `Wall_Direction` is incremented or decremented by 90 degrees each time the robot makes a turn. This forces the limitation of working in an orthogonal world. Another reason for working in an orthogonal world will be discussed at the end of this chapter.

As mentioned earlier, MML controls the robot's heading by telling it to track a line. Yamabico knows the heading of the line it is supposed to be tracking. If it is tracking a line that is 90 degrees, and it is told it is really heading 80 degrees, it knows to correct to the left 10 degrees.

Equation 1 will be used as the basis to calculate the correction on the other two walls. When tracking wall 1 or wall 3, the algorithm is a bit more complex. Figure 4 shows that had the robot tracking 10 degrees to the left, α returned a value of 80. The algorithm should tell the robot it is not heading 0 degrees but 10 degrees. Using equation 1 would return a value of -80 degrees. A 10 degree error to the right would return a value of 80 degrees. Both of these are very far off the value needed to correct the heading. From Equation 1, α must be subtracted from the `Wall_Direction`. If α is less than 0, subtract 90 degrees from Equation 1 to get the actual heading. If α is greater than 0, add 90 degrees from Equation 1 to get the actual heading. That leads to the equation

```
Actual_Heading = Wall_Direction - alpha + Wall_Variable,        (2)
```

where `Actual_Heading` is the heading the robot is actually tracking, `Wall_Direction` is the direction of the wall and `Wall_Variable` is a value that toggles between 0 and 90. Initially, `Wall_Variable` has a value of 90 degrees. Each time the robot makes either a left or right turn, `Wall_Variable` toggles between 90 and

0. If `Wall_Variable` is 90 it is changed to 0. If it is 0, it is changed 90. Refer to Appendix B for a more thorough explanation and a proof of this algorithm.

C. MOVING ABOUT IN THE ROOM

The behavior of moving about the room while determining the direction of the wall can be broken down into many subtasks. We will begin with the most basic tasks and will work up to the more difficult tasks.

1. Parallel the Wall

The robot must sense the wall and maintain its movement parallel to it. Refer to the previous section for an explanation of this algorithm.

2. Maintains Proper Distance from the Wall

Initially, this was thought to be important. However, after programming Yamabico, it did not seem to be critical. The author felt that it was more important to minimize the number of heading changes than to attempt to get the robot at a precise distance from the wall.

The distance can be roughly obtained by altering the distance, which triggers a turn. If a larger distance from the wall is desired, the robot should turn left a farther distance from the wall. Through experimentation, it was more desirable to keep the robot close to the wall so the right turns would be made more consistently. This is discussed in more detail in Chapter IV.

3. Recognizes Obstacle to the Front and Turns Left

For the purposes of this thesis, the robot will maintain the wall on the right side of the robot. The right side sonar will be used to follow the wall, the front sonar will be used

to detect approaching obstacles or walls, and the left side sonar will be used to find objects in the middle of the room. The methods described will just as easily work tracking the wall on the left side of the robot.

As the robot travels down the wall, it must look to its front and take notice if there is an obstacle to the front. If there is, the robot must turn left the proper distance from the wall to be set up at the proper distance from the next wall.

4. Recognizes the Wall that it is following Ends and Turns Right

As the robot travels down the wall, it must notice if a right turn is required. If the right sonar suddenly gets very large, the robot must recognize the fact that the wall has ended. It must then wait for the precise moment to make the right turn so it is set up at the proper distance from the wall as it travels on its mission.

5. Recognizes the Original Starting Point

The robot must recognize when it has followed the walls of the room around back to the starting point. It is not possible to simply have the robot stop when its heading exceeds 2π. A room with a complex shape such as a spiral could cause the heading of the robot to exceed 20π. The best solution will be to calculate the difference in distance between Yamabico's starting location and its current location. When the distance is less than a preset threshold, the initial portion of the program should end. However, the robot must have exceeded a distance threshold before this check can be made.

6. Processes Wall Segments

After Yamabico returns to the starting point, the sonar segments for the left sonar and the right sonar must be processed to determine if there are any unexplored objects in the room. As the robot is moving about the room, the left sonar will be activated. This

will allow the robot to sense objects about four meters to the left of the robot. Add to that four meters, the four meters as the robot passes along the opposite wall and the meter or so between the robot and the wall it is scanning gives a total width of 10 meters or almost 33 feet. If the room has a width of more than 10 meters, it would be possible to calculate that from the wall segments. The robot would be required to determine if one or more passes down the middle of the room would be required to cover the area that has not been seen by the sonar. Since 10 meters is large enough to accommodate most rooms in a building, this thesis will not bother with determining if the room is too large for one pass.

After Yamabico completes one circuit of the room, many wall segments may define one wall. These segments must be grouped together to form one segment for each wall. If the segments are relatively close to each other and almost parallel, the segments are merged into one.

If the room is so narrow that the left side sonar sees the wall on the opposite side of the room, it will create these segments not knowing it is the other wall and that the right side sonar will soon see that wall. All the segments in the left sonar segments will be matched to the segments in the right sonar segment list. When a match is made of the left sonar segment, that segment from the left side is deleted from the list. After all the segments in the list from the left-side sonar have been matched, any remaining segments that are still not matched must be an object in the middle of the room.

If the room were very large, the robot would follow the wall around the room until it saw the unpaired segment with the left sonar. At that point, the robot would turn left until it saw the segment in the front sonar. When it is at the proper distance, it would turn left to face the right sonar to the wall. The robot would make one loop around that

object and stop again at the place it started from. It would then turn left to join the outer wall and travel around the walls until it saw the next unmatched left sonar segment. After the last left sonar segment was complete, the robot will continue around the room until it returned to its original starting point. Note: the task discussed in this paragraph was not implemented in this thesis.

D. OUTPUTTING THE MAP

After the robot circumnavigates the room, it must present the data that represents the walls in the room in a readable fashion. One assumption of this thesis is that the room and all the objects in it are orthogonal. Due to inaccuracies in sonar, the line segments that were calculated by the least squares fitting may not be perfectly orthogonal. The wall segments may have to be adjusted so that they are actually orthogonal. The conclusion will mention any errors that are caused by this normalization.

There are many different ways a map could be represented. Using MML, the best way to represent the lines is by the values calculated when performing the line fitting. The values are the start and end points, α, r and the length of the line. By maintaining the calculated values, any sonar reading could be compared with each segment to determine which segment it is seeing. Comparing the sonar points to known walls, the robot can use the intersection of two known segments for odometry corrections and navigation.

E. MAP MAKING IN A NON-ORTHOGONAL WORLD

A simulator was developed in Lisp to determine if automated cartography could be accomplished in a non-orthogonal world. It was determined that even more errors would be introduced into the result.

When Yamabico wall-follows in an orthogonal world, it is possible to correct the minor heading errors. If it is known that the wall is 0, 90, 180, or 270 degrees, and Yamabico believes that a wall it is following is not heading one of those directions, the robot's heading must be wrong. By comparing the difference between the robot's heading and the direction of the wall, a correction to the heading can be determined. In MML, the command `setRobotConfigImm` tells Yamabico what its current position and orientation are. If Yamabico is told it is heading 5 degrees and it is supposed to be tracking a line of 0 degrees, it knows to correct to the left 5 degrees.

If Yamabico is placed in a non-orthogonal world, heading corrections can not be made, they can only be adjusted. Suppose the robot has a heading of 0 degrees and the wall direction is calculated to be 5 degrees. It is not known if the wall direction is 5 degrees or the wall direction is 0 degrees and the robots internal odometry is off by 5 degrees.

In order to get very precise positioning, a robot must have a link to a known position. When the robot starts, it is told where it is in the world. As soon as it starts to move, slight errors begin to accumulate and the exact position of the robot is no longer known. The exact position can be calculated if there is a link to a known position. That link may be perfect odometry, which is impossible. More likely, the link to an exact position could be through the use of GPS or sensors placed in locations known to the robot and located throughout the robot's environment.

Although heading can not be corrected, it is possible for the robot to follow the walls around the room. This author believes that the errors from the lack of heading correction will lead to increasing errors and may or may not return a useable map as

output. Chapter IV discusses the problems encountered when the heading can not be corrected.

III. THE SOFTWARE

A. MODEL BASED MOBILE-ROBOT LANGUAGE (MML)

In MML, the autonomous rigid-body vehicle defines its current state by a transformation that consists of its x- and y-position in the global coordinate system, its heading and the amount of curvature the vehicle is moving. For example, if a robot is at a position (10, 100) and moving in the y-direction on the global coordinate system in a straight line, the transformation would be $(10, 100, \frac{\pi}{2}, 0)$. The autonomous robot defaults the global origin to be its initial position. The robot can also calculate the global origin if it is given its current location and direction. A benefit of MML is that objects such as lines and curves are also defined by the same four parameters. A circle with its center at the origin and a radius of 50 centimeters would be defined as (0, 50, 0, -0.02). To define the curve, it is only required that any point on that line be defined for the robot to track it. The robot does not even need to go to the point that defines the line. It could merely touch the edge of it on the other side and it will still work.

Yamabico is capable of tracking straight lines or curved lines. It is aware of the line it is currently supposed to be following and it continually checks to see if it is time to turn towards the next line.

Take the most basic example of tracking from one straight line to another straight line. A variable called sigma, σ, tells the robot how sharp to make the turn. σ is the inverse of the radius, so if σ is very small, the radius of the turn will be very large. A large turning radius would be necessary for a robot that had a high center of gravity or operated at high speeds.

There is an optimal point at which to leave the current line to arrive at the next line. As you can see in Figure 5, if the robot leaves the current line too soon, it will swing to the right before it makes its turn to the left. This is undesirable. If the robot leaves too late, it will cross the path it is supposed to follow and eventually tracks it. This is also undesirable. There is an optimal point between these two examples that will cause the

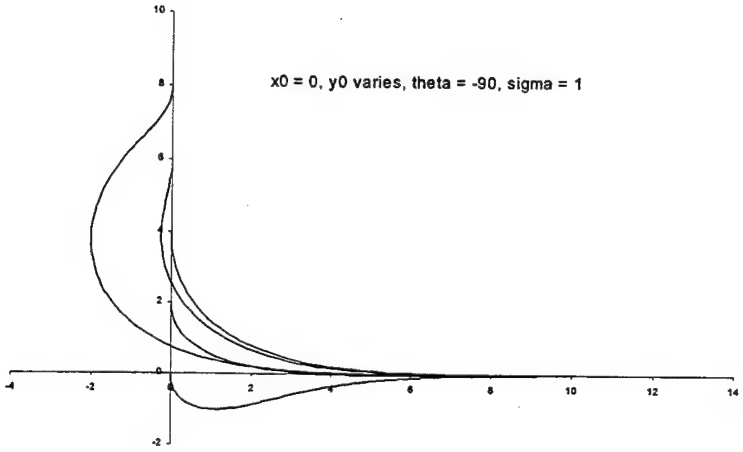

Figure 5. Example of Turning Capability.

robot to meet up exactly with the next line. There is also an optimal departure point for line to circle tracking and circle to circle tracking.

The robot maintains its current position by the use of internal odometry. If the robot uses sonar to detect obstacles, it can calculate the position of that obstacle in the robot's frame of reference or in the global reference.

As the robot moves through its environment, it is continually updating its current position. It can keep a log of its position during each experiment. It is also capable of keeping the global coordinates of each object the sonars see.

Another feature of MML is the line-fitting algorithm. It uses the Generalized Least-Squares Fitting. Least squares fitting is a method of determining the best fitting line for a given set of points by minimizing the total distance from the points to the fitted line. This method is very straightforward and simple to understand.

If the sonar log is activated, each time that sonar receives data, it is compared to the current line segment to determine if it is within a certain threshold. If it is, the point is added to the list of points. If it is not, a new line segment is created. Much information can be determined from the least squares fitting, including: the end points of the fitted line, the angle of the fitted line in the global plane, the distance from the closest point on the fitted line to the origin and the correctness of the fit.

One of the many benefits to using MML is the vast library of functions that can be called to accomplish the objective. If you wanted the robot to start from the origin at a heading of 0 degrees, and follow the outline of a star with each side 200 centimeters long, the code would look like this.

```
1   void user () {

2     sideLength = 200.0;
3     init = defineConfig(0.0, 0.0, 0.0, 0.0);
4     line0 = defineConfig(0.0, 0.0, 0.0, 0.0);
5     line1 = defineConfig(sideLength, 0.0, 4.0*PI/5.0, 0.0);

6     setRobotConfigImm(init);

7     for (i=0; i<=4; i++) {
8       track(line0);
9       line0 = compose(&line0, &line1);
10    }
11    trackS(init);

12    return;
13  }
```

The function `defineConfig` defines the transformation mentioned earlier in this chapter. `init` defines the robot's current position, heading, and curvature. `setRobotConfigImm(init)` tells the robot what its current configuration is. It should be noted that `init` and `line0` are could be used interchangeably. They were defined separately for clarity. `line1` defines the next line that needs to be tracked. It says from the current location, travel a distance of `sideLength`, then $\frac{4\pi}{5}$ radians or left 135 degrees.

The `for` loop is used to cycle through the program so that it will draw all five sides of the star. The function `compose` is the heart of MML. That is what gives it the power to control the robot. Line 8 assigns the variable `line0` the next line to be tracked. `trackS(init)` tells the robot to stop when it gets back to the location defined by `init`.

Aside from declaring the variables, nothing else is needed to program the robot. This code is saved in a file called 'user.c'. As MML is compiled and executed, MML initializes the odometry, sonars, and other operations of the robot then calls the function `user`. After the function `user` is complete, housekeeping is performed and the robot is shut down. If logging functions were initialized, the robot would display a message notifying the user to prepare the robot for downloading of the files.

There are many other functions of MML. Many of them will be described as they are used in the code.

B. USE OF MML IN THIS THESIS

This section will describe how the algorithms are implemented in the C Programming Language. Refer to Appendix A for a listing of the file 'user.c'.

1. Declaration and Initialization of the Variables

The first section of the program is the declaration and initialization of the variables. The important variables are:

`CONFIGURATION` variables. These define the initial configuration of the robot, which is at global position 100, 100. Its heading is 0 degrees and it will start traveling in a straight line. `Configuration left` is a line that is defined as (20, 0, $\frac{\pi}{2}$, 0). When composed with the robot's current configuration, that tells the robot to imagine a straight line 20 centimeters to the front that is heading 90 degrees to the left of the current heading. A similar configuration is defined for right turns.

- `wallVar` and `wallDir` are defined in Chapter II so they will not be discussed in detail here.
- `pingcount` and `delayCount` were created to improve the performance of the robot. `pingcount` is the number of data points that make up the current wall segment. If there are only a few data points, the true characteristics of the object may not yet be acquired. If there are say 20 or more points in the segment, it is more likely that the data segment will be accurate. `delayCount` is used as a program delay. After the program makes a heading correction, a delay is begun to prevent the program from making another heading correction until the first has taken effect. `delayCount` is a counter that is set to zero after a heading correction has been made. Each time the program cycles through the program loop, `delayCount` is incremented by one. When it reaches a threshold value, another heading correction can be made. A timed delay loop would not be a good design decision because while the 'user.c' program was counting the delay, the robot could run into a wall.

- `sigma` is a variable used by the motion control portion of MML. This variable defines the sharpness of the turns from one line to the next. The higher the value of sigma, the smoother the motion.
- `flag` is a boolean that is initialized to true. When the robot travels a distance of more than 100 centimeters, it is set to false. The use of this variable will be described in more detail later in the chapter.

The next portion of 'user.c' is the initialization of variables.

- `EnableSonar(S090);` activates the sonars at the 0-, 90-, 180-, and 270-position of the robot. Since the 000, 090 and 270 sonars are used in this thesis, any one of the three sonars could be activated.
- `EnableLinearFitting(S090);` tells Yamabico to use generalized least squares fitting to the sonar points that are returned from S090. These segments area used to determine where the walls are. `EnableLinearFitting(S270)` is also activated.
- `InitSonarLog();` tells Yamabico to create files to hold sonar data.
- `MapLog();, SonarLog();` and `MotionLog();` tells the robot what parameters to use to create the Map, Sonar and Motion log files.
- `setLinearVelImm();` sets the forward velocity of the robot. This is not critical to this exercise but could be a concern for more delicate maneuvers.
- `setSigmaImm();` sets sigma for Yamabico.
- `setRobotConfigImm();` initializes the configuration of the robot.
- `track(init);` tells Yamabico to begin motion along the path defined by the configuration variable `init`.

2. Circumnavigate the Room and Gather Information

The next section of the code is the portion that travels around the room, gathering sonar data to create the map.

This portion begins with a while statement that tests for the exit condition. There are two conditions that will allow it to continue looping; 1) if `flag` is true OR 2) if the robot is greater than 100 centimeters from the starting point. Since `flag` is initialized to true, that conditional will pass until `flag` is changed to false. During the course of the loop, a check is made to determine how far the robot is from the starting point. If it is greater than 100 centimeters, `flag` is set to false. Now when the code checks the flag, it is false, but the second part of the conditional is true. When the robot returns to within 100 centimeters from the starting point, both parts of the conditional will be false and the while loop is exited.

Inside the loop, the distance is obtained from each of the sonars by the command `dist090 = Sonar(S090);` where `S090` is the sonar you wish to get data from and `dist090` is the variable you want the information going into. Next, `pingcount` is assigned the number of points in the current segment.

The first check determines if there is an obstacle to the front of the robot. If there is, `wallDir` is incremented by 90 degrees. `sigma` is set to a low value so a sharp turn is made. The sonar interrupts are disabled. The sonar interrupt is not intuitive and it doesn't seem necessary, but if the interrupts are not disabled, Yamabico will not function properly while making left turns. The author believes this is a quirk in the robot's hardware.

Yamabico's current configuration is composed with the left configuration and Yamabico is told to track the new line. The program is delayed so the robot can get on track before the sonar interrupts are re-activated. The value of `wallVar` is toggled from 0 to HPI or from HPI to 0, and `delayCount` is set to a value that allows a heading correction to be made immediately.

The next check determines if the distance from the right sonar to the closest obstacle has exceeded a certain threshold. If it has, a right turn is required. First, `wallDir` is decremented by HPI. Since the Sonar090 is required to determine if the robot is abeam the next wall, the sonar interrupts are disabled and the sonar log for 090 is paused so extraneous segments are not formed. Yamabico's current configuration is composed with the right configuration to get the next line to track. The sonar interrupts are re-activated and the distance returned by the right sonar is checked. If the distance is greater than some threshold, the robot must not have reached the corner yet and the sonar logging must wait. A short while loop is entered until the distance returned by the right sonar is within the proper limits.

After the threshold for the right sonar is met, the program is paused for a very short time to allow the robot to be abeam the corner of the walls. Next `wallVar` is updated and `delayCount` is modified, like in the left-hand turn.

If neither a left-hand turn nor a right-hand turn is required, a check is made of the proper heading. If the current line segment has the minimum number of sonar points and the `delayCount` has elapsed, a correction to the heading can be made. The values for the current segment are obtained. The actual heading is calculated and corrected, and

`delayCount` is set to 0. This variable prevents the heading from being corrected so frequently that it is unstable. This is the final if statement is the sequential if-statements.

The next events are `delayCount` is incremented by 1 and `flag` is checked to see if it should be changed. The conditional in the while loop checks `flag` and the distance between the robots current position and the starting point. If the distance exceeds a threshold, `flag` is set to 0. This will have the effect of causing the while loop to terminate when the robot returns to a point that is within the threshold value.

3. Generate the Map

The next objective is to generate the map. During the course of the wall following, the line segments for the left and right sonars were placed into separate arrays. These segments must be cleaned up so they can be compared to one another.

Two points define each segment. The first two points are compared to the next two points to see if they could be part of the same wall. If the orientation of the wall is generally the same, both of the points that define the second segment are within a certain distance to the line created by the first line segment and they are adjacent in the segment list, they are considered to be the same wall. These four points now make up the segment. The next two points are compared to this new segment. This process continues until all segments have been compared to all adjacent segments for the left and right array of segments.

When this process is completed, the segments from the modified left and right sonar segment lists must be compared. The objective of the next part is to find any segment seen by the left sonar that was not seen by the right sonar. This would imply that the left sonar saw an object in the middle of the room. If the left sonar saw an object and

the right sonar saw the same object, this means that the sonar saw the wall from across the room.

When the comparison of the segments is complete, if there are any segments in the left sonar segment list that have not been matched to the right sonar segment list, they must be investigated. Yamabico must follow the walls around until the unmatched sonar segment comes into view of the left sonar. At that point, Yamabico must turn left and head toward the segment. Then the standard wall-following program is run. The robot follows that object around until it returns to where it started from. It determines if it has seen any other unmatched wall segments. If it has, it returns to the outside wall and follows it around to the starting point. If it has not, it also follows the wall around but only until it sees the next unmatched wall segment. It then repeats the process until all unmatched wall segments have been investigated and the robot is back to its original starting point. Note: this has not been implemented in this thesis.

Even though Yamabico is run in an orthogonal world, due to the many errors described in Chapter IV, the line segments generated by the line fitting algorithm will not be perfectly orthogonal. In order to be useable to other robots for navigation and/or path planning, the segments will be adjusted to appear more like the actual room. The segments will be adjusted so they are all orthogonal to each other. The segments will also be extended so they meet the adjacent segments at right angles to form corners.

After the robot completes one lap around the room, the array of right segments will have line segments defining the outside walls and the left segments will define the objects in the middle of the room. Each segment array is processed identically and separately.

First the lines are straightened. If the line is generally oriented in the 0- to 180-degree direction, the y-values of the start and end points are averaged. If the line is oriented in the 90- to 270- direction, x-values are averaged.

Next, the lines must be extended so they form proper corners. Each line is compared to every other line in the array. Each line is compared to every other line in the array. The line that is chosen to form the corner with the first line will have the start point nearest the end point of the first line and be perpendicular to the first line. Then the intersection of the two lines is calculated. The end of the first line and the beginning of the second line are assigned the value of the intersection. Every line must be compared to every other line because the left sonar may have adjacent segments in its array but they may not be physically adjacent to each other. After all this is done, the entire map is shifted so the left most wall is on the y-axis and the lower most wall is on the x-axis for aesthetic purposes.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CHARACTERISTICS OF THE HARDWARE

A. CHARACTERISTICS OF SONARS

Ultrasonic sensors or sonars are widely used on mobile robots for sensing the outside world. The predominant reason for using sonars is because they are inexpensive and easy to use. But, they also have many characteristics that may lead to errors. These characteristics can be grouped into three areas:

1) Environmental or atmospheric conditions,
2) Beam width and the reflection of the sonar away from the receiver.
3) Texture of the target.

1. Range Errors Due to the Atmosphere

Ultrasonic sensors measure distance by timing the flight of the sound wave. The time is measured from when the sound leaves the sonar until it returns. The distance is determined by multiplying half the time by the speed of sound. The speed of sound in air is $v = (331 + .610t)$ m/s where t is the temperature in degrees Celsius. If the room temperature rose from 60°F to 80°F, speed of sound would increase from 342.2 m/s to 349.0 m/s. If the sonar was calibrated for a temperature of 60°F, an obstacle 200 cm from the sonar would be calculated at 195.1 cm. This is an error of 4.9 cm or about 2.5%. If the sonar were calibrated for 70°F, the error would be cut in half.

It is the author's belief that the environment causes only small errors when using sonar to perform navigation and obstacle avoidance.

2. Sonar Beam Width and Target – Sonar Axis Angle

Another characteristic of sonar that can give erroneous ranges is the width of the sonar beam. The waves created by dropping a pebble in a smooth pool of water are similar to sound waves emanating from a source. The sound travels outward from the source in an arc. If an object is anywhere within the arc, the sound may bounce off that object and return to the receiver. Since the receiver can not know where the object is in the arc, it must assume it is in the direct line with the axis of the emitter. Figure 6 shows a sonar return coming off the wall at point A, but the sonar energy returned is believed to be from point B. This leads to distance errors. This error can be minimized if the robot's sonar maintains a perpendicular orientation to the wall.

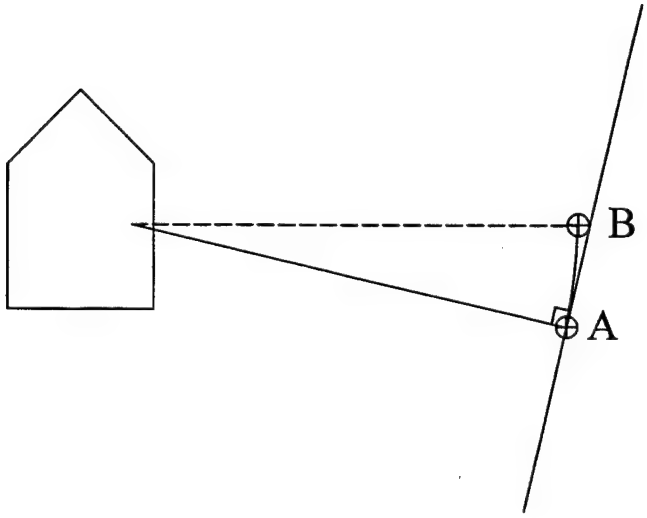

Figure 6. Beam Width Error.

Figure 7 shows experiment results that were obtained by placing Yamabico in one position and moving the "wall" to different distances and angles. The experiment was conducted to measure the width of the sonar beam. A ½" particleboard was placed perpendicular to the sonar axis at a distance of 50 cm. The sonar distance was read. The wall was then pivoted at increments of 5 degrees at the point where the axis of the sonar

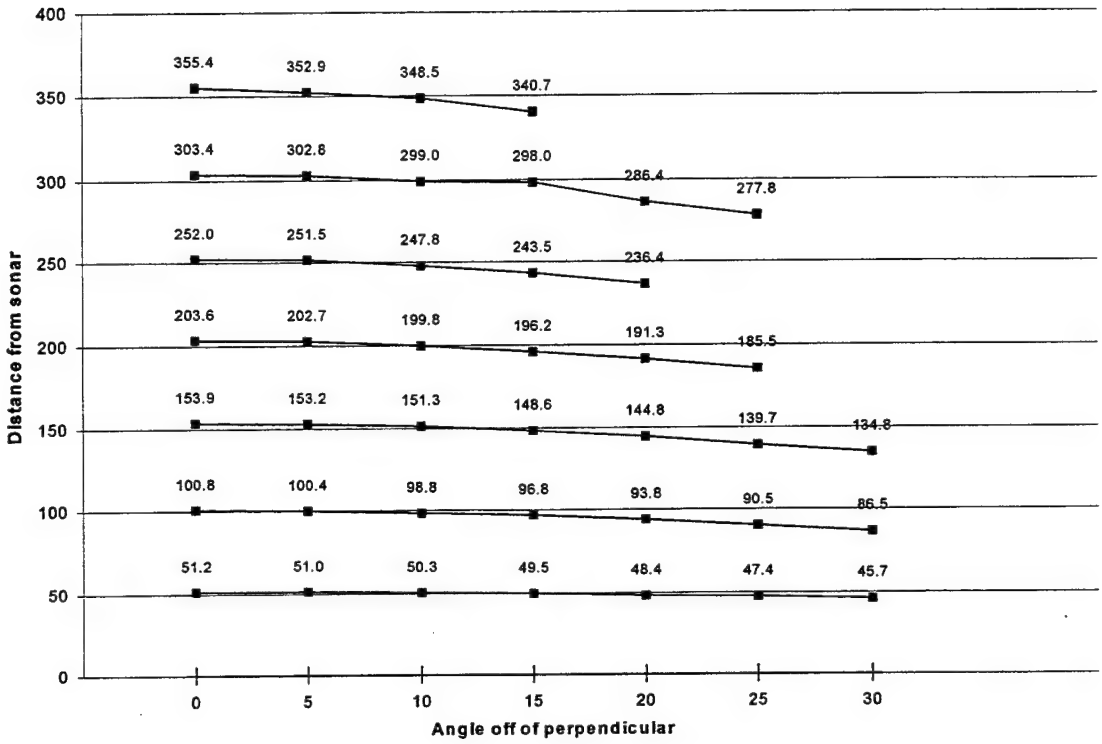

Figure 7. Experiment with Sonar at Different Angles in Relation to the Wall.

intersected with the wall. This experiment was done at distances from 50 cm to 350 cm at increments of 50 cm. When the wall was 150 centimeters or less to the sonar, a steady return was received when the "wall" was as much as 30 degrees off perpendicular to the sonar axis. As the distance from the wall to the sonar increased, returns were received when the wall was 15 degrees or less off perpendicular. It should also be noted that as the angle of the wall moved away from perpendicular to the sonar axis, the distance to the target was perceived to be closer and closer.

The beam width can also cause a return to be seen in an open area. If the corner of a wall falls within in the width of the beam, a return may be seen which would extend the wall farther than it actually is. Figure 8 shows sonar returns from the robot passing an obstacle. The object was artificially drawn to show its location. In this experiment, the

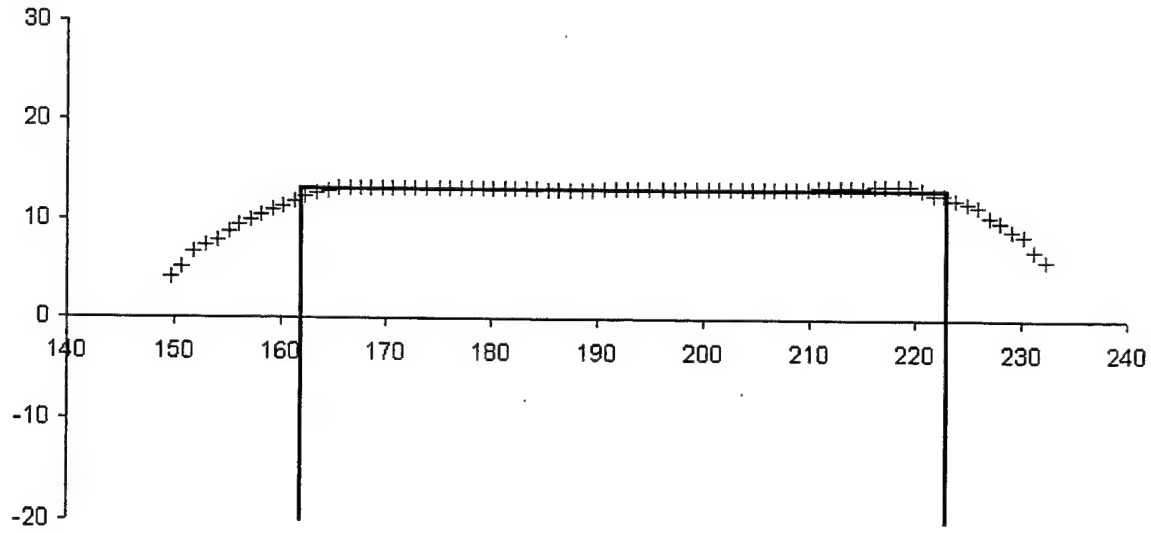

Figure 8. Effects of Convex Corner on Sonar Returns.

robot paralleled the wall with the sonar at a distance of approximately 65 cm. The sonar begins to receive returns from the corner before it comes abeam the corner. If the robot is using that sonar to parallel the wall, and there are only a few returns that make up that segment, it may cause the robot start paralleling the wall that is rotated left of the actual wall. As the robot passes the corner on the other end of the wall, the wall will also appear to be extended and curved at the end. Since many data points define the wall by the time it gets the other end, the robot is not likely to turn right and follow the curve of the sonar returns. However, this could cause the robot to turn later than would be expected.

Another example of the inaccuracy of sonar can be seen in Figure 9. Yamabico was placed in the center of a box with the size of 366 cm by 142 cm. The robot was programmed to rotate for one complete rotation with one sonar measuring distances and calculating the global position of the return. The locations where the sonar returns were seen are designated by '+'. The curves on the top and the bottom of the box in the figure are caused by the sonar beam width. This phenomenon is described in the section A-2 of

this chapter and a drawing is shown in Figure 6. The curves on the left and right sides of the figure are also caused by the same phenomenon. The apparent extension of the curved lines outside of the box are caused by the sound energy bouncing off one wall and being returned by another wall. Since the particle board that was used is a good reflector of the sound energy, the sound returned may have reflected off several walls before it bounced back to the receiver.

From the experiments conducted for this thesis, the author has determined that the total width of the beam for the sonars on Yamabico have a total beam width of 60°, or 30° on each side of the sonar axis.

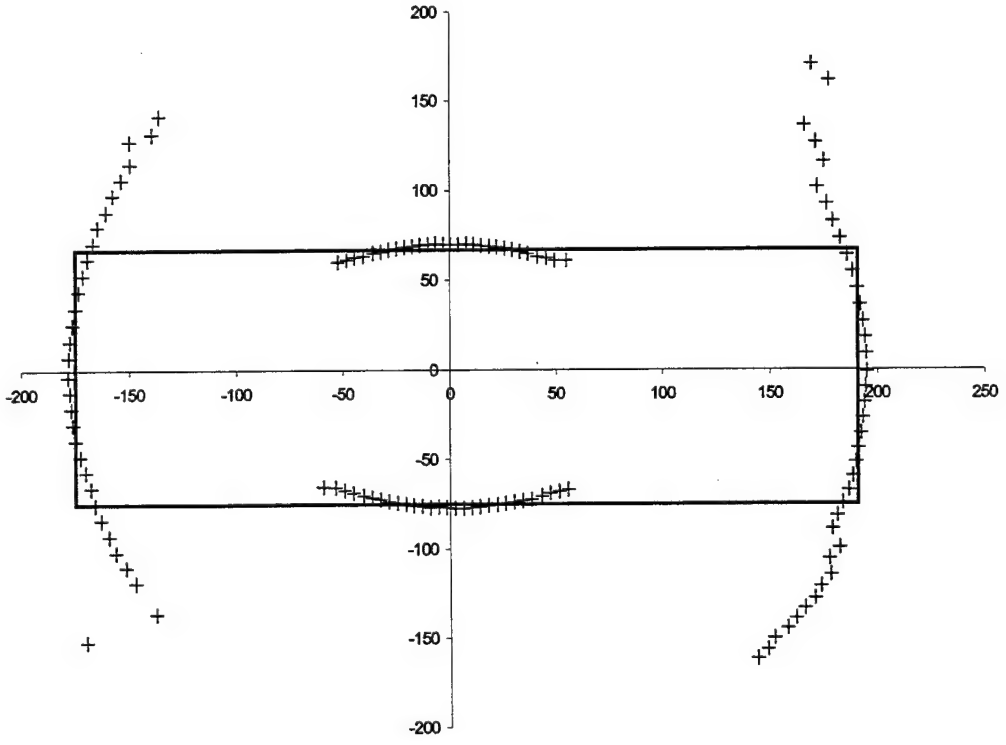

Figure 9. Sonar Returns by Rotation Robot 360 Degrees.

3. Texture of the Target

All of the experimental data derived during the investigation of this thesis was using 1/2-inch thick particleboard. It is flat and very good at reflecting the sound energy. Other objects are not as good at reflecting the sound. A curtain is an example of a common household item that would not reflect the sound well. The material of the curtain absorbs some of the sound energy. In addition to absorbing the sonar, the texture of the object can also affect the distance that is measure to it. The distance to an object can be in error as much as 7% due to the composition of the target [Lochner, 1994]. The shape and/or orientation of the object could also affect the sonar returns. Much of the sound energy would be trapped in the curtain's folds and would not be reflected back to the receiver.

B. WHEEL SLIPPAGE AND SHAFT ENCODERS

Wheel slippage can be caused by a number of factors. If the floor is very slippery, has low coefficient of friction, the robot may become displaced in a direction other the direction of travel or the wheels may spin or skid. If the robot turns a corner too sharply for the speed it is traveling, or accelerate/decelerate to rapidly, the wheels could slide and the robot would not know how far it has been displaced from its intended path.

An analogy can be made between a robot with no ability to update its position and a person with no ability to see. Both know their original position by what is told to them. Both have a limited reach to determine where obstacles are. If the robot slides in a direction that is perpendicular to the direction of travel, it will not notice that it has slid. And it can definitely not know how far it slid. The same with a blind man. If he is placed

on an escalator, he may notice that he is moving in a general direction but he will not know how far.

If the floor is very bumpy, errors can also accumulate. If the robot is traveling in a straight line and one wheel hits a bump, the heading of the vehicle and the distance traveled will accumulate errors. Even if the robot has a magnetic compass to determine heading changes, the odometry will still be off a small amount that can not be detected by just the heading change. If the blind man is walking over uneven terrain and trying to measure distance by his pace count, the unequal footing will cause the blind man to quickly accumulate errors because this pace will not be consistent.

The size of the tire's footprint is another cause of errors that is associated with the wheel. As the robot makes a turn, no matter what the size of the footprint, it must rotate in precisely one spot. Whether the tire is trying to pivot in place or the robot is in motion and making a very gradual turn, there will always be a pivot point. If the footprint is large, it would be impossible to determine where the pivot point would be. For this reason, it is better to have the smallest footprint possible. This means the tires must be very narrow and very firm. If the tire were to be made of steel, the footprint would be very small, but ability of the robot to maintain traction would suffer. If the robot had tires that were soft so traction was very good, the size of the footprint would be larger. A compromise must be made between a high coefficient of friction and on the type of tire to make a small foot print.

Another cause of small amount of error associated with the mechanical workings of the shaft encoder. Compared to other sources of errors, these are very small systematic errors and may possibly be removed by adjusting this error source through software

programming [Borenstien 96]. Borenstien devised a method to determine the systematic errors in the odometry [Borenstien 96]. This would still not overcome the non-systematic errors.

Figure 10 and Figure 11 show the effect of being able to make heading correction by taking a reference off the orientation of the wall. The line with the small squares is the path that Yamabico believes it followed during the experiment. The path shows the center of movement for the robot. The small diamonds are the global position of the sonar returns that were received using the sonar on the right side of the robot. In Figure 10, you can see that the walls appear to continually move in towards the robot's path. As Yamabico continues to make the circuits, the error continues to accumulate. All the while, Yamabico believes it is still following a path that are increments of one-half pi. This is most likely cause of the errors are due to wheel slippage and systematic error caused by inaccuracies in the shaft encoder. As Yamabico followed the wall around in Figure 11, it continually made heading corrections to prevent it from coming into contact with the walls.

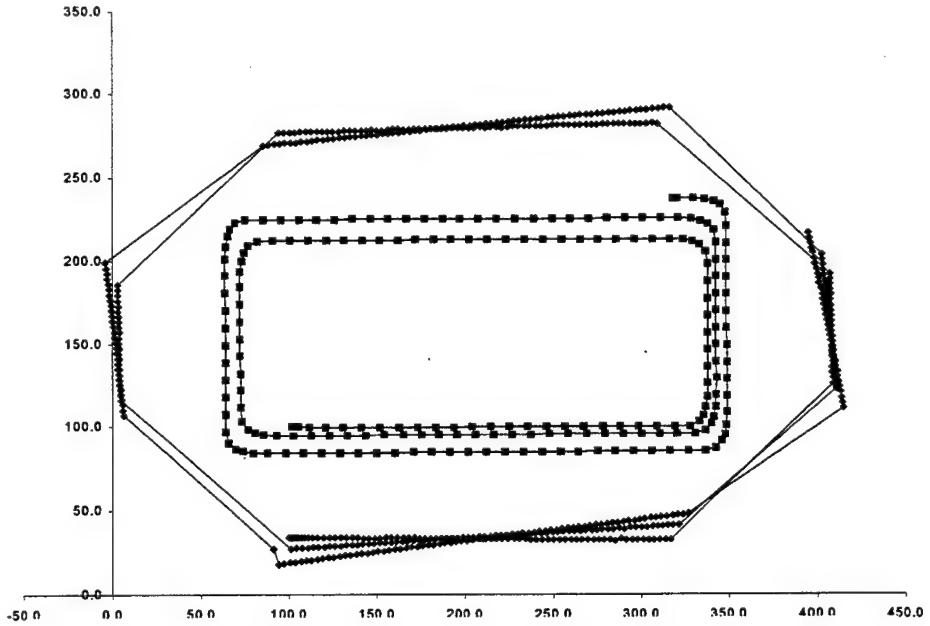

Figure 10. Wall Following without Heading Correction.

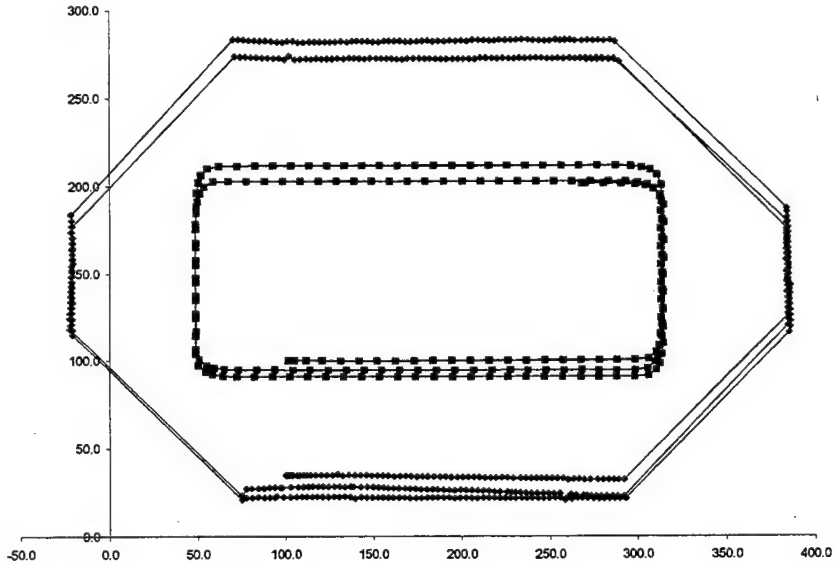

Figure 11. Wall following with Heading Correction.

THIS PAGE INTENTIONALLY LEFT BLANK

V. THESIS RESULTS AND CONCLUSION

A. THESIS RESULTS

To determine if the results achieved by this thesis were good or not, depends on the purpose for creating the map. If the map were to be used by a robot for navigation, the map would be more than sufficient. Given the map, a robot could use it to easily navigate through a room. The robot could use its sonar to update its current position by using corners of the room, or obstacles in the room as landmarks for odometry correction.

The map created by Yamabico shown Figure 12 is very close to the actual floor plan. The largest error was along the y-axis which had an error of 5 cm. The actual distance was 438 cm so that gives an error of about 1 percent. The solid line is output

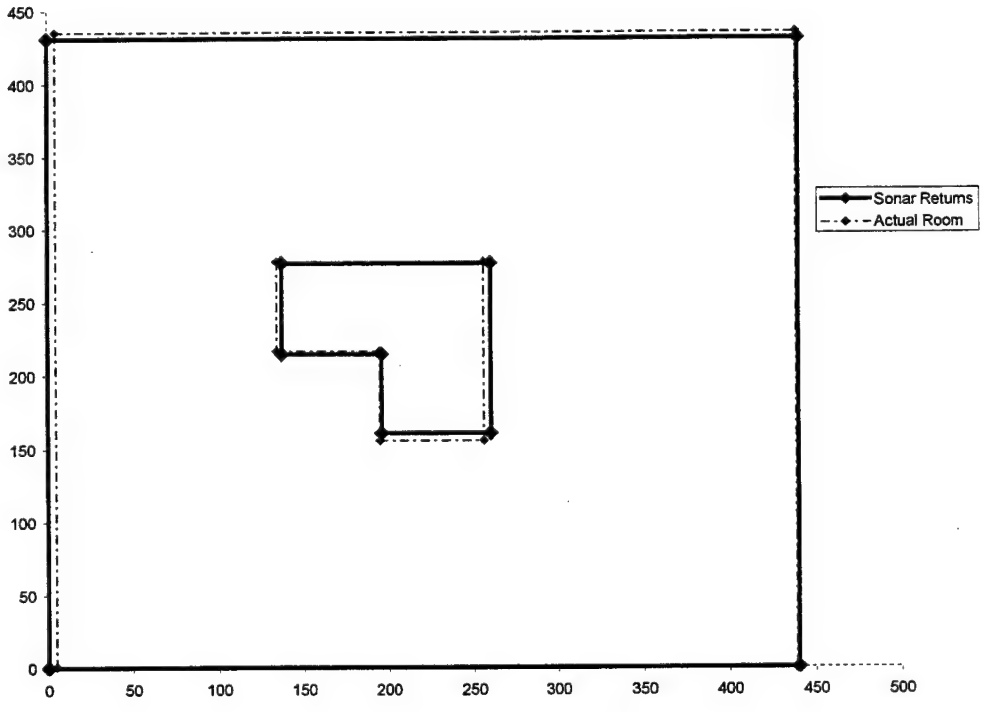

Figure 12. Map 1 Created by Yamabico.

from the map program in Yamabico. The dashed line is the room's actual measurements. The error in Figure 13 was slightly larger. The largest error was again along the y-axis. This time it was 11 cm or about 2.5 percent. Plotting the map generated by Yamabico

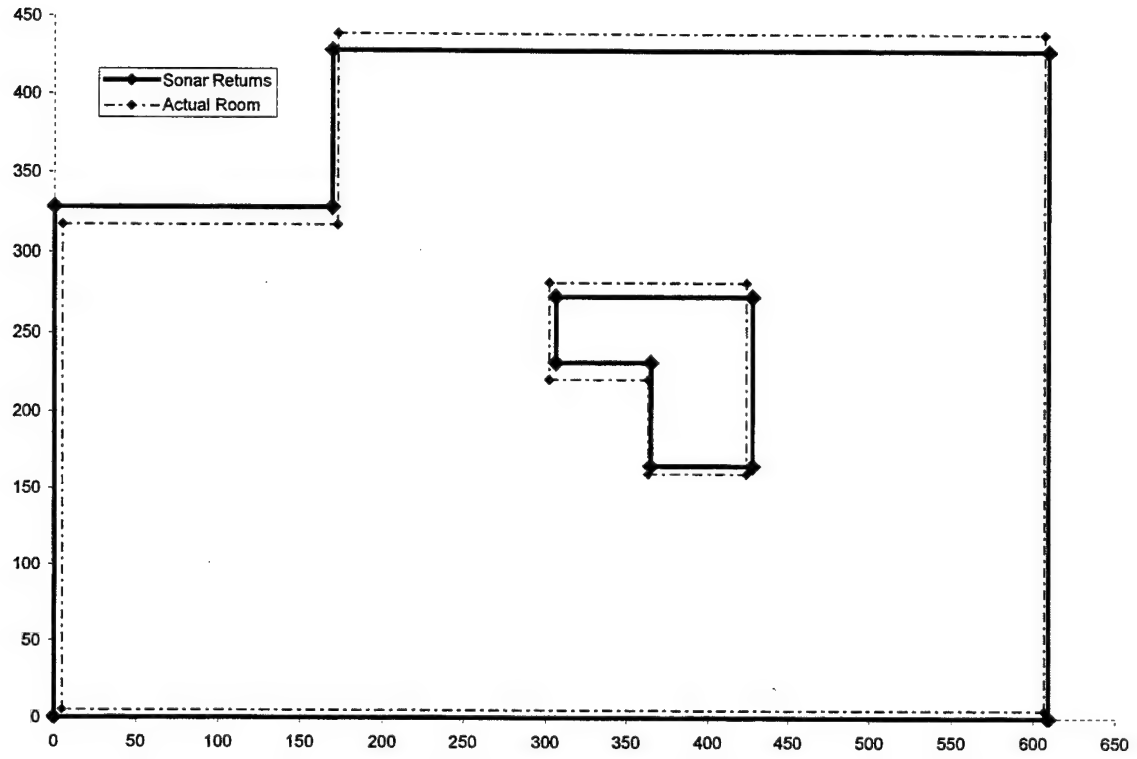

Figure 13. Map B created by Yamabico.

created these figures. Then the actual floor plan of the room was placed in a best-fit position.

It should be mentioned that these results were obtained in a clutter-free room. There were no objects in the room other than the "walls". If there were other objects, the line segments would not be as "clean" and many more errors would have been created.

The program created for this thesis was limited to perform in an orthogonal world. If the program were modified so that it would control a robot in a non-orthogonal world, this author believes the error would be much greater. With the current orthogonal

restriction, when the robot determines that the heading it is following is not the same direction as the wall it is following, the robot knows that the wall is correct and the robot is wrong. This continually corrects the heading throughout the entire experiment. If the world were not orthogonal, the robot would not know if its heading had drifted or the wall had turned. The only option would be to assume that the wall had turned. Soon this would lead to very large heading errors.

B. CONCLUSION

The results produced by the mobile robot were better than expected. The map is accurate enough to be used by Yamabico or another robot to navigate by in the room. The map could be used for path planning or obstacle avoidance.

This program could be enhanced so that a robot could explore through doorways and more complex environments. Another enhancement would be to track the obstacles in the middle of the room with the right sonar as described in Chapter II. A follow on thesis could be to use the map created by Yamabico to perform path planning.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. USER FILES

This appendix lists the code used to create the maps for this thesis. The file user.c is the heart of the program that makes Yamabico do wall following. The file follow.c contains the administrative functions that were written specifically for this thesis. It should be noted that the wall-following was the most concise part of the coding. The manipulation of the line segments to create the map was a majority of the code.

A. USER.C

```
/*****************************************************************/
/* File: user.c                                                  */
/* Name: Mark Merrell                                            */
/* Date: 3 March 1999                                            */
/* Description: This file is called by the MML main program. This    */
/* file is used to perform wall following that will produce a map of */
/* the room. Many of the administrative functions are in follow.c    */
/*****************************************************************/

#include "user.h"
#include "seqcmd.h"
#include "math.h"
#include "queue.h"
#include "follow.h"

void user()
{

   CONFIGURATION const init = defineConfig(100.0, 100.0, 0.0, 0.0);
   CONFIGURATION left = defineConfig(20.0, 0.0, HPI, 0.0);
   CONFIGURATION right = defineConfig(25.0, 0.0, -HPI, 0.0);
   CONFIGURATION current, actualPath, path;

   SEGMENT_RES * CurrSegment;

   double dist000, dist090;
   int const delayTime = 40;
   int pingcount = 0, minpingcount = 20;
   int selection, delayCount = delayTime;
   double wallVar = HPI, wallDir = 0;
   double sigma = 15;
   int flag = 1;
   double startx = init.Posit.X, starty = init.Posit.Y;
```

```
printf("\nContinuous Curvature Motion Control: Version 450 ");
selection = GetInt();
if (selection != 0) {
   return;
}

/* initalize the robot's environment */

EnableSonar(S090);
EnableLinearFitting(S090);
EnableLinearFitting(S270);

InitSonarlog();
setSonarLogFrequency(S090, 10);
setSonarLogFrequency(S270, 10);

SonarLog(0, 50000, S090, SONAR_SEGMENT);
SonarLog(0, 50000, S090, SONAR_GLOBAL);
SonarLog(0, 50000, S270, SONAR_SEGMENT);
SonarLog(0, 50000, S270, SONAR_GLOBAL);
MotionLog(0,50,0);
MapLog("map.log",1,0);

setLinVelImm(20.0);
setSigmaImm(sigma);

setRobotConfigImm(init);
current = getRobotConfig();
track(init);

while (flag || (320 < (sqrt (Power((current.Posit.Y - starty), 2) +
                             Power((current.Posit.X - startx), 2))))) {

   dist090 = Sonar(S090);
   waitMS(50);
   dist000 = Sonar(S000);
   printf(".");

   pingcount = GetNumPoints(S090);

   if ((dist000 < 55) && (dist000 > 20)) {
      DisableSonarInterrupts();
      printf("\nTURNING LEFT");
      printf("\ndistance 000 = %4.2f", dist000);
      wallDir += HPI;
      setSigmaImm(3);
      current = getRobotConfig();
      path = compose(&current, &left);
      path.Kappa = 0.0;
      track(path);

      waitSec(3);
      if (wallVar == 0) {
         wallVar = HPI;
      } else {
```

```
      wallVar = 0;
   }
   setSigmaImm(sigma);
   delayCount =  delayTime;
   dist000 = 999.0;
   EnableSonarInterrupts();

} else if (dist090 > 100) {
   DisableSonarInterrupts();

   printf("\nTURNING RIGHT ");
   wallDir -= HPI;
   PauseSonarLog(S090);
   setSigmaImm(3);
   current = getRobotConfig();
   path = compose(&current, &right);
   path.Kappa = 0.0;
   track(path);

   waitSec(3);
   EnableSonarInterrupts();
   dist090 = Sonar(S090);
   while (dist090 > 60) {
      waitMS(500);
      dist090 = Sonar(S090);
   }

   waitSec(1);
   ResumeSonarLog(S090);

   if (wallVar == 0) {
      wallVar = HPI;
   } else {
      wallVar = 0;
   }
   setSigmaImm(sigma);
   delayCount =  delayTime;

} else if ((pingcount > minpingcount) && (delayCount > delayTime)) {

   CurrSegment = GetCurrentSegment(S090);
   current = getRobotConfig();
   actualPath = current;

   if (CurrSegment->alpha < 0) {
      wallVar = -fabs(wallVar);
   } else {
      wallVar = +fabs(wallVar);
   }

   actualPath.Theta = wallDir - CurrSegment->alpha + wallVar;
   setRobotConfigImm(actualPath);
   delayCount = 0;
}
delayCount = delayCount + 1;
```

```
        current = getRobotConfig();
        if (flag && (310 < (sqrt (Power((current.Posit.Y - starty),2) +
                                  Power((current.Posit.X - startx),2))))) {
            flag = 0;
        }
    }

    DisableLinearFitting(S090);
    DisableLinearFitting(S270);
    DisableSonar(S000);
    DisableSonar(S030);
    DisableSonar(S090);
    DisableSonar(S270);
    FlushBuffer();
    stopImm();

    GenerateMap();

    return;
}
```

B. FOLLOW.C

```
/*********************************************************************/
/* File: follow.c                                                    */
/* Name: Mark Merrell                                                */
/* Date: 3 March 1999                                                */
/* Description: This file is called by user.c to perform             */
/* administrative functions. One of the many functions in this file  */
/* is GenerateMap. This function takes two arrays of SEGMENT_WORK's  */
/* and performs a Union on them. The results are put in the output   */
/* file map.log.                                                     */
/*********************************************************************/

#include <math.h>
#include "sonar.h"
#include "trace.h"
#include "follow.h"
#include "stdiosys.h"

IOhandle    MapHandle;
static int  Enabled;


/*********************************************************************
PURPOSE:    Initialize the map.log file
PARAMETERS: Filename, Tracing frequency and buffer size
RETURNS:    void
COMMENTS:   Creates the file "map.log". This file is written to at the
            end of GenerateMap, the last function in this file.
            "map.log" will contain the final output from this program,
            a complete map of the robot's environment.
*********************************************************************/

void
MapLog(char *filename, int Frequency, int bufSize)
{
   if (bufSize <= 0)
      bufSize = 10000;     /* if not specified, use the default */

   if (filename == NULL)
      filename = "map.log";

   if (Frequency <= 0)
      Frequency = 1;

   MapHandle = IOopen(filename, bufSize, Frequency);

   if (MapHandle == EOF)  {
      printf("\nError initializing map logging file %s\n", filename);
      return;
   }
   Enabled = 1;
}
```

```
/*********************************************************************
PURPOSE:    Determines a line segment is still at initial values
PARAMETERS: A line segment
RETURNS:    1 or 0, for true or false
COMMENTS:   The line segments in the array are initialized to all
            zeros. This function checks to see if the start point and
            end point are both at the origin. If they are, the line
            segments have not been written to and they are null.
*********************************************************************/

int
null(SEGMENT_WORK segment) {
   int answer;

   if (segment.seg.start.X == 0.000 &&
       segment.seg.start.Y == 0.000 &&
       segment.seg.end.X == 0.000 &&
       segment.seg.end.Y == 0.000) {

      answer = 1;
   } else {
      answer = 0;
   }

   return answer;
}


/*********************************************************************
PURPOSE:    Determines if line segment no longer has its initial values
PARAMETERS: A line segment
RETURNS:    1 or 0 for true or false
COMMENTS:   This function calls null and takes the inverse of the value
            that is returned.
*********************************************************************/

int
not_null(SEGMENT_WORK segment) {

   return !(null(segment));
}


/*********************************************************************
PURPOSE:    Determine the distance between two points
PARAMETERS: Two points
RETURNS:    a double
COMMENTS:
*********************************************************************/

double
Distance (POINT Point1, POINT Point2) {
   return (sqrt (Power((Point1.X - Point2.X),2) +
                 (Power((Point1.Y - Point2.Y),2))));
}
```

```
/*********************************************************************
PURPOSE:     Initialize an array to zero
PARAMETERS:  The point to an array of line segments
RETURNS:     void
COMMENTS:    Initializes the start point and end point of each line
             segment in the array. This allows a check to be made to
             find the end of the array.
*********************************************************************/

void Initialize_Array(SEGMENT_WORK * SegArray) {

   int loopCount = 0;

   while (loopCount < 100) {
     SegArray[loopCount].seg.start.X = 0.000;
     SegArray[loopCount].seg.start.Y = 0.000;
     SegArray[loopCount].seg.end.X   = 0.000;
     SegArray[loopCount++].seg.end.Y = 0.000;
   }

   return;
}


/*********************************************************************
PURPOSE:     Finds the closest line segment to the end point of Line1
PARAMETERS:  A line segment and the array of line segments
RETURNS:     an index to the closest line segment
COMMENTS:    Finds the closest line segment to the end point of Line1
             that passes the following criteria:
                   - within 300 centimeters of the end point of Line1
                   - the line segment must have an orientation within 45
                     degrees of Line1
                   - the line segment can not be the same line as Line1
*********************************************************************/

int FindClosest(SEGMENT_WORK Line1, SEGMENT_WORK * SegArray) {
   int counter = 0, answer = -1;
   double short_distance = 999999.99, distance;

   while (not_null(SegArray[counter])) {

      distance = Distance(SegArray[counter].seg.start, Line1.seg.end);
      if ((distance < short_distance) &&
          (distance < 300) &&
          (fabs(Line1.seg.alpha - SegArray[counter].seg.alpha) > .78) &&
          (SegArray[counter].seg.end.X != Line1.seg.end.X) &&
          (SegArray[counter].seg.end.Y != Line1.seg.end.Y)) {
         answer = counter;
         short_distance = distance;
      }
      counter++;
   }
   return answer;
}
```

```
/*********************************************************************
PURPOSE:    Fills in the segment array with information about the line
segments
PARAMETERS: An array of line segments
RETURNS:    void
COMMENTS:   This function calls the findLine function for each line
            segment in the line segment array. Flagpoint is a flag
            value to let the function findLine know that this line
            segment has only two points.
*********************************************************************/

void Calculate_Seg_Array_Data(SEGMENT_WORK * SegArray) {

   POINT flagPoint;
   int counter;

   flagPoint.X = 9999.9;
   flagPoint.Y = 9999.9;

   counter = 0;
   while (not_null(SegArray[counter])) {
      SegArray[counter] = findLine(SegArray[counter], flagPoint);
      counter++;
   }
   return;
}


/*********************************************************************
PURPOSE:    Finds the intersection of two line segments
PARAMETERS: Two line segments
RETURNS:    a POINT
COMMENTS:   This function finds the intersection point of two line
            segments if each of the line segments were extended to
            infinity in both directions.
*********************************************************************/

POINT FindIntersection (SEGMENT_WORK Line1, SEGMENT_WORK Line2) {
   POINT Point1 = Line1.seg.start;
   POINT Point2 = Line2.seg.start;
   POINT answer;
   double theta1 = Line1.seg.alpha + HPI;
   double theta2 = Line2.seg.alpha + HPI;
   double temp1 = Point1.X * sin(theta1) - Point1.Y * cos(theta1);
   double temp2 = Point2.X * sin(theta2) - Point2.Y * cos(theta2);
   answer.X = (temp2 * cos(theta1) - temp1 * cos(theta2)) /
                  sin (theta2 - theta1);
   answer.Y = (temp2 * sin(theta1) - temp1 * sin(theta2)) /
                  sin (theta2 - theta1);
   return answer;
}
```

```
/*********************************************************************
PURPOSE:    Determines the distance between a point and a line segment
PARAMETERS: A POINT and a line segment
RETURNS:    a double representing the distance
COMMENTS:
*********************************************************************/

double DistancePointLine (SEGMENT_WORK LineSegment, POINT xyLocation) {

  double a = LineSegment.seg.start.X, b = LineSegment.seg.start.Y;

  double theta = LineSegment.seg.alpha + HPI;
  double x = xyLocation.X, y = xyLocation.Y;
  double part1 = (y - b) * cos(theta);
  double part2 = (x - a) * sin(theta);

  return abs(part1 - part2); /* SHOULD THIS BE fabs!?!?!*/

}


/*********************************************************************
PURPOSE:    Fits a point into the line segment
PARAMETERS: A line segment and a POINT
RETURNS:    A line segment
COMMENTS:   Adds a point to the line segment then calculates the new
            attributes for the line segment. If the function is called
            with a line segment that has only two points and no
            attributes, a flag should be passed in to identify the fact
            that the line does not have any attributes yet. The flap
            should be a POINT with the x and y value of 9999.9
*********************************************************************/

SEGMENT_WORK findLine (SEGMENT_WORK Segment, POINT p) {

   double    m00, m10, m01, m20, m11, m02;
   double    alpha, r, length;
   double    mux, muy, mm20,mm11, mm02;
   double    x, y, startx, starty, endx, endy;
   double    distanceSP, distanceEP;

   if (p.X == 9999.9 && p.Y == 9999.9) { /* only using start and end */

      startx = Segment.seg.start.X;
      starty = Segment.seg.start.Y;
      endx = Segment.seg.end.X;
      endy = Segment.seg.end.Y;

      m00 = Segment.m00 = 2.0;
      m10 = Segment.m10 = startx + endx;
      m01 = Segment.m01 = starty + endy;
      m20 = Segment.m20 = SQR(startx) + SQR(endx);
      m11 = Segment.m11 = startx * starty + endx * endy;
      m02 = Segment.m02 = SQR(starty) + SQR(endy);
```

```
    } else {                        /* adding a point to the segment */

       m00 = Segment.m00 += 1.0;
       m10 = Segment.m10 += p.X;
       m01 = Segment.m01 += p.Y;
       m20 = Segment.m20 += SQR(p.X);
       m11 = Segment.m11 += p.X * p.Y;
       m02 = Segment.m02 += SQR(p.Y);
       distanceSP = sqrt(SQR(Segment.seg.start.X - p.X) +
                         SQR(Segment.seg.start.Y - p.Y));
       distanceEP = sqrt(SQR(Segment.seg.end.X - p.X) +
                         SQR(Segment.seg.end.Y - p.Y));
       if ((distanceSP > Segment.seg.length) &&
           (distanceSP > distanceEP)) {
          Segment.seg.end = p;
       }

       if ((distanceEP > Segment.seg.length) &&
           (distanceEP > distanceSP)) {
          Segment.seg.start = p;
       }
    }

    mux = m10 / m00;
    muy = m01 / m00;
    mm20 = m20 - SQR(m10) / m00;
    mm11 = m11 - m10 * m01 / m00;
    mm02 = m02 - SQR(m01) / m00;
    y = -2.0 * mm11;
    x = mm02 - mm20;

    if (y == 0.0 && x == 0.0) {
       alpha = 0.0;
    } else {
       alpha = atan2(y, x) / 2.0;
    }

    r = mux * cos(alpha) + muy * sin(alpha);
    length = sqrt(SQR(Segment.seg.start.X - Segment.seg.end.X) +
                  SQR(Segment.seg.start.Y - Segment.seg.end.Y));

    Segment.seg.alpha = alpha;
    Segment.seg.r = r;
    Segment.seg.length = length;

  return Segment;
}
```

```
/*********************************************************************
PURPOSE:    Displays the array of segments to the user
PARAMETERS: An array of line segments
RETURNS:    void
COMMENTS:   This displays the array of line segments to the terminal so
            the user can debug the code.
*********************************************************************/

void displayArray(SEGMENT_WORK * SegArray) {

   int counter = 0;

   while (not_null(SegArray[counter])) {
      printf("\n%4.2f %4.2f %4.2f %4.2f %4.2f",
                              SegArray[counter].seg.start.X,
                              SegArray[counter].seg.start.Y,
                              SegArray[counter].seg.r,
                              SegArray[counter].seg.length,
                              SegArray[counter].seg.alpha);
      printf("\n%4.2f %4.2f\n", SegArray[counter].seg.end.X,
                                SegArray[counter].seg.end.Y);
      counter++;
    }

    return;
}

/*********************************************************************
PURPOSE:    Generates the map from line segments generated by MML
PARAMETERS: void
RETURNS:    The map in the file "map.log"
COMMENTS:   Reads the line segments from the 090 and 270 sonars.
            Determines if two separate line segments can be merged into
            one line. Determines if any line segments from 270 segments
            can be 'absorbed' by the 090 line segments. Straightens the
            line segments so they are all orthogonal to the origin.
            Extends the line segments so that physically adjacent line
            segments are joined.
*********************************************************************/

void GenerateMap() {

   SEGMENT_WORK LCombineSeg[100], RCombineSeg[100];
   POINT Point1, Point2;
   int found_one;
   int scounter, ccounter;
   int lcount, ucount, rcount;
   double lateral_distance = 30.0;
   int nextLine;
   double x_adjust = 9999.99, y_adjust = 9999.99;

   Initialize_Array(RCombineSeg);
   Initialize_Array(LCombineSeg);
```

```
/* Calculate alpha, r, and length, for each line segment */

Calculate_Seg_Array_Data(SegmentArray[7]);
Calculate_Seg_Array_Data(SegmentArray[5]);

/* Combines the line segments. If a wall is made up of several */
/* segments, this part will make them into one segment         */

scounter = 0;
ccounter = 0;

while (not_null(SegmentArray[7][scounter])) {
   if (SegmentArray[7][scounter].seg.length < 20) {
       scounter++;
   } else if (null(RCombineSeg[ccounter])) {
       RCombineSeg[ccounter] = SegmentArray[7][scounter];
       scounter++;
   } else if ((DistancePointLine (RCombineSeg[ccounter],
                SegmentArray[7][scounter].seg.start) < lateral_distance) &&
              (DistancePointLine (RCombineSeg[ccounter],
                SegmentArray[7][scounter].seg.end) < lateral_distance) &&
              ((fabs(RCombineSeg[ccounter].seg.alpha) -
                fabs(SegmentArray[7][scounter].seg.alpha)) < PI/4)) {
      RCombineSeg[ccounter] = findLine (RCombineSeg[ccounter],
                                SegmentArray[7][scounter].seg.start);
      RCombineSeg[ccounter] = findLine (RCombineSeg[ccounter],
                                  SegmentArray[7][scounter].seg.end);
      scounter++;
   } else {
      ccounter++;
      RCombineSeg[ccounter] = SegmentArray[7][scounter];
      scounter++;
   }
}

if ((DistancePointLine (RCombineSeg[0],
       RCombineSeg[ccounter].seg.start) < lateral_distance) &&
    (DistancePointLine (RCombineSeg[0],
       RCombineSeg[ccounter].seg.end) < lateral_distance)) {
      RCombineSeg[0] = findLine (RCombineSeg[0],
                                   RCombineSeg[ccounter].seg.start);
      RCombineSeg[0] = findLine (RCombineSeg[0],
                                   RCombineSeg[ccounter].seg.end);
      RCombineSeg[ccounter].seg.start.X = 0.000;
      RCombineSeg[ccounter].seg.start.Y = 0.000;
      RCombineSeg[ccounter].seg.end.X = 0.000;
      RCombineSeg[ccounter].seg.end.Y = 0.000;
}

scounter = 0;
ccounter = 0;

while (not_null(SegmentArray[5][scounter])) {
   if (SegmentArray[5][scounter].seg.length < 20) {
       scounter++;
```

```
      } else if (null(LCombineSeg[ccounter])) {
          LCombineSeg[ccounter] = SegmentArray[5][scounter];
          scounter++;
      } else if ((DistancePointLine (LCombineSeg[ccounter],
                    SegmentArray[5][scounter].seg.start) < lateral_distance) &&
                  (DistancePointLine (RCombineSeg[ccounter],
                    SegmentArray[5][scounter].seg.end) < lateral_distance) &&
                  ((fabs(LCombineSeg[ccounter].seg.alpha) -
                     fabs(SegmentArray[5][scounter].seg.alpha)) < PI/4)) {
         LCombineSeg[ccounter] = findLine (LCombineSeg[ccounter],
                                   SegmentArray[5][scounter].seg.start);
         LCombineSeg[ccounter] = findLine (LCombineSeg[ccounter],
                                     SegmentArray[5][scounter].seg.end);
         scounter++;
      } else {
         ccounter++;
         LCombineSeg[ccounter] = SegmentArray[5][scounter];
         scounter++;
      }
   }


   if ((DistancePointLine (LCombineSeg[0],
          LCombineSeg[ccounter].seg.start) < lateral_distance) &&
       (DistancePointLine (LCombineSeg[0],
          LCombineSeg[ccounter].seg.end) < lateral_distance)) {
         LCombineSeg[0] = findLine (LCombineSeg[0],
                                    LCombineSeg[ccounter].seg.start);
         LCombineSeg[0] = findLine (LCombineSeg[0],
                                    LCombineSeg[ccounter].seg.end);
         LCombineSeg[ccounter].seg.start.X = 0.000;
         LCombineSeg[ccounter].seg.start.Y = 0.000;
         LCombineSeg[ccounter].seg.end.X = 0.000;
         LCombineSeg[ccounter].seg.end.Y = 0.000;
   }

/* This part is supposed to compare the right and left segments. Any */
/*   segment that is in the left segment that can not be merged with */
/*   one of the right segments must be an obstacle in the middle of  */
/*   the room. The result is put back into the left segments.        */

   lcount = 0; /* left count      */
   ucount = 0; /* unmatched count */
   rcount = 0; /* right count     */

   while (not_null(LCombineSeg[lcount])) {
       found_one = 0;
       rcount = 0;
       while (not_null(RCombineSeg[rcount])) {
          if ((DistancePointLine (RCombineSeg[rcount],
                   LCombineSeg[lcount].seg.start) < lateral_distance) &&
              (DistancePointLine (RCombineSeg[rcount],
                   LCombineSeg[lcount].seg.end) < lateral_distance)) {
             found_one = 1; /* Current segment can be eliminated */
          }
```

```
            rcount++;
        }

        if (found_one != 1) {
            LCombineSeg[ucount] = LCombineSeg[lcount];
            ucount++;
        }
        if ((ucount - 1) != lcount) {
            LCombineSeg[lcount].seg.start.X = 0.000;
            LCombineSeg[lcount].seg.start.Y = 0.000;
            LCombineSeg[lcount].seg.end.X = 0.000;
            LCombineSeg[lcount].seg.end.Y = 0.000;
        }
        lcount++;
    }

/* This code makes all the segments orthogonal to the origin. It */
/*   also determines the smallest y value and smallest x value.  */
/*   These values are   used to move lower left corner of the    */
/*   map to the origin.                                          */

    ccounter = 0;
    while (not_null(RCombineSeg[ccounter])) {
        if (fabs (RCombineSeg[ccounter].seg.alpha) < 0.78) {
            Point1.X = RCombineSeg[ccounter].seg.start.X;
            Point2.X = RCombineSeg[ccounter].seg.end.X;
            RCombineSeg[ccounter].seg.start.X = (Point1.X + Point2.X) / 2.0;
            RCombineSeg[ccounter].seg.end.X = (Point1.X + Point2.X) / 2.0;
            RCombineSeg[ccounter].seg.alpha = 0.0;
        } else {
            Point1.Y = RCombineSeg[ccounter].seg.start.Y;
            Point2.Y = RCombineSeg[ccounter].seg.end.Y;
            RCombineSeg[ccounter].seg.start.Y = (Point1.Y + Point2.Y) / 2.0;
            RCombineSeg[ccounter].seg.end.Y = (Point1.Y + Point2.Y) / 2.0;
            RCombineSeg[ccounter].seg.alpha = HPI;
        }
        if (RCombineSeg[ccounter].seg.start.X < x_adjust) {
            x_adjust = RCombineSeg[ccounter].seg.start.X;
        }
        if (RCombineSeg[ccounter].seg.end.X < x_adjust) {
            x_adjust = RCombineSeg[ccounter].seg.end.X;
        }
        if (RCombineSeg[ccounter].seg.start.Y < y_adjust) {
            y_adjust = RCombineSeg[ccounter].seg.start.Y;
        }
        if (RCombineSeg[ccounter].seg.end.Y < y_adjust) {
            y_adjust = RCombineSeg[ccounter].seg.end.Y;
        }
        ccounter++;
    }

    ccounter = 0;
    while (not_null(LCombineSeg[ccounter])) {
        Point1.X = LCombineSeg[ccounter].seg.start.X;
        Point2.X = LCombineSeg[ccounter].seg.end.X;
```

```
      if (fabs (LCombineSeg[ccounter].seg.alpha) <  0.78) {
         Point1.X = LCombineSeg[ccounter].seg.start.X;
         Point2.X = LCombineSeg[ccounter].seg.end.X;
         LCombineSeg[ccounter].seg.start.X = (Point1.X + Point2.X) / 2.0;
         LCombineSeg[ccounter].seg.end.X = (Point1.X + Point2.X) / 2.0;
         LCombineSeg[ccounter].seg.alpha = 0.0;

      } else {
         Point1.Y = LCombineSeg[ccounter].seg.start.Y;
         Point2.Y = LCombineSeg[ccounter].seg.end.Y;
         LCombineSeg[ccounter].seg.start.Y = (Point1.Y + Point2.Y) / 2.0;
         LCombineSeg[ccounter].seg.end.Y = (Point1.Y + Point2.Y) / 2.0;
         LCombineSeg[ccounter].seg.alpha = HPI;
      }
      if (LCombineSeg[ccounter].seg.start.X < x_adjust) {
         x_adjust = LCombineSeg[ccounter].seg.start.X;
      }
      if (LCombineSeg[ccounter].seg.end.X < x_adjust) {
         x_adjust = LCombineSeg[ccounter].seg.end.X;
      }
      if (LCombineSeg[ccounter].seg.start.Y < y_adjust) {
         y_adjust = LCombineSeg[ccounter].seg.start.Y;
      }
      if (LCombineSeg[ccounter].seg.end.Y < y_adjust) {
         y_adjust = LCombineSeg[ccounter].seg.end.Y;
      }
      ccounter++;
   }

   Calculate_Seg_Array_Data(LCombineSeg);
   Calculate_Seg_Array_Data(RCombineSeg);

/* This section extends the line segments so that physically */
/*   adjacent line segments meet at a corner.                */

   ccounter = 0;
   while (not_null(RCombineSeg[ccounter])) {
      nextLine = FindClosest(RCombineSeg[ccounter], RCombineSeg);
      if (nextLine != -1) {
         Point1 = FindIntersection(RCombineSeg[ccounter],
                                                RCombineSeg[nextLine]);
         RCombineSeg[ccounter].seg.end = Point1;
         RCombineSeg[nextLine].seg.start = Point1;
      }
      ccounter++;
   }

   ccounter = 0;
   while (not_null(LCombineSeg[ccounter])) {

      nextLine = FindClosest(LCombineSeg[ccounter], LCombineSeg);
      if (nextLine != -1) {
         Point1 = FindIntersection(LCombineSeg[ccounter],
                                                LCombineSeg[nextLine]);
         LCombineSeg[ccounter].seg.end = Point1;
```

```
            LCombineSeg[nextLine].seg.start = Point1;
        }
        ccounter++;
    }

/* This section writes the result of the map to the file map.log   */

    ccounter = 0;
    while (not_null(LCombineSeg[ccounter])) {

        LCombineSeg[ccounter].seg.start.X =
                            LCombineSeg[ccounter].seg.start.X - x_adjust;
        LCombineSeg[ccounter].seg.end.X   =
                            LCombineSeg[ccounter].seg.end.X   - x_adjust;
        LCombineSeg[ccounter].seg.start.Y =
                            LCombineSeg[ccounter].seg.start.Y - y_adjust;
        LCombineSeg[ccounter].seg.end.Y   =
                            LCombineSeg[ccounter].seg.end.Y   - y_adjust;

        IOprintf(MapHandle, "\n%f %f", LCombineSeg[ccounter].seg.start.X,
                                       LCombineSeg[ccounter].seg.start.Y);
        IOprintf(MapHandle, "\n%f %f\n", LCombineSeg[ccounter].seg.end.X,
                                         LCombineSeg[ccounter].seg.end.Y);
        ccounter++;
    }

    ccounter = 0;
    while (not_null(RCombineSeg[ccounter])) {

        RCombineSeg[ccounter].seg.start.X =
                            RCombineSeg[ccounter].seg.start.X - x_adjust;
        RCombineSeg[ccounter].seg.end.X   =
                            RCombineSeg[ccounter].seg.end.X   - x_adjust;
        RCombineSeg[ccounter].seg.start.Y =
                            RCombineSeg[ccounter].seg.start.Y - y_adjust;
        RCombineSeg[ccounter].seg.end.Y   =
                            RCombineSeg[ccounter].seg.end.Y   - y_adjust;

        IOprintf(MapHandle, "\n%f %f", RCombineSeg[ccounter].seg.start.X,
                                       RCombineSeg[ccounter].seg.start.Y);
        IOprintf(MapHandle, "\n%f %f\n", RCombineSeg[ccounter].seg.end.X,
                                         RCombineSeg[ccounter].seg.end.Y);
        ccounter++;
    }

    return;
}
```

APPENDIX B. ANALYSIS OF HEADING CORRECTION ALGORITHM

In order for the heading correction algorithm to be valid, I had to prove that it would provide the correct heading correction for any possible α. Below I have listed the facts that provide the basis for my algorithm.

FACTS:

- `Wall_Variable` is initialized to 90.
- `Wall_Direction` is initialized to 0.
- When a turn is made to the left, 90 is added to the `Wall_Direction`. 90 is subtracted from `Wall_Direction` if the turn is to the right.
- When a turn is made, if the `Wall_Variable` is 90 it is changed to 0. If `Wall_Variable` is 0, it is changed to 90.
- if $\alpha < 0$
  `Wall_Variable` is made negative;
  else
  `Wall_Variable` is made positive;
  end if;

My algorithm to calculate the robot's actual heading:

```
Actual_Heading = Wall_Direction - alpha + Wall_Variable
```

Wall 1. This is the wall the robot will initially be tracking. If the robot is on track the `Actual_Heading` will equal to 0.

```
Wall_Direction = 0
alpha = -90   // robot is perpendicular to the wall
Wall_Variable = -90
Actual_Heading = 0 - (-90) + (-90) = 0

Wall_Direction = 0
-90 < alpha < 0
```

```
Wall_Variable = -90
Actual_Heading = 0 - (0) + (-90) as alpha approaches 0,
      Actual_Heading approaches -90
Actual_Heading = 0 - (-90) + (-90) as alpha approaches -90,
      Actual_Heading approaches 0

Wall_Direction = 0
0 < alpha < 90
Wall_Variable = 90
Actual_Heading = 0 - (0) + (90) as alpha approaches 0,
      Actual_Heading approaches 90
Actual_Heading = 0 - (90) + (90)  as alpha approaches 90,
      Actual_Heading approaches 0
```

Wall 2. After the robot makes a left turn, it will be on Wall 2. Any wall is classified as Wall 2 if the robot is tracking it 90 degrees to the left of the original heading. If the robot is on track, the `Actual_Heading` will be 90.

```
Wall_Direction = 90
alpha = 0
Wall_Variable = 0
Actual_Heading = 90 - (0) + 0 = 90

Wall_Direction = 90
-90 < alpha < 0
Wall_Variable = 0
Actual_Heading = 90 - (0) + (0) as alpha approaches 0,
      Actual_Heading approaches 90
Actual_Heading = 90 - (-90) + (0) as alpha approaches -90,
      Actual_Heading approaches 180

Wall_Direction = 90
0 < alpha < 90
Wall_Variable = 0
Actual_Heading = 90 - (0) + (0) as alpha approaches 0,
      Actual_Heading approaches 90
Actual_Heading = 90 - (90) + (0) as alpha approaches 90,
      Actual_Heading approaches 0
```

Wall 3. Any wall is classified as Wall 3 if the robot is tracking it 180 degrees off the original heading. If the robot is on track, the `Actual_Heading` will be 180.

```
Wall_Direction = 180
alpha = -90
Wall_Variable = -90
Actual_Heading = 180 - (-90) + (-90) = 180

Wall_Direction = 180
-90 < alpha < 0
Wall_Variable = -90
Actual_Heading = 180 - (0) + (-90) as alpha approaches 0,
     Actual_Heading approaches 90
Actual_Heading = 180 - (-90) + (-90) as alpha approaches
     -90, Actual_Heading approaches 180

Wall_Direction = 180
0 < alpha < 90
Wall_Variable = 90
Actual_Heading = 180 - (0) + (90) as alpha approaches 0,
     Actual_Heading approaches 270
Actual_Heading = 180 - (90) + (90) as alpha approaches 90,
     Actual_Heading approaches 180
```

Wall 4. Any wall is classified as Wall 4 if the robot is tracking it 270 degrees off the original heading. If the robot is on track, the `Actual_Heading` will be 270.

```
Wall_Direction = 270
alpha = 0
Wall_Variable = 0
Actual_Heading = 270 - (0) + 0 = 270

Wall_Direction = 270
-90 < alpha < 0
Wall_Variable = 0
Actual_Heading = 270 - (0) + (0) as alpha approaches 0,
     Actual_Heading approaches 270
Actual_Heading = 270 - (-90) + (0) as alpha approaches -90,
     Actual_Heading approaches 360

Wall_Direction = 270
0 < alpha < 90
Wall_Variable = 0
Actual_Heading = 270 - (0) + (0) as alpha approaches 0,
     Actual_Heading approaches 270
Actual_Heading = 270 - (90) + (0) as alpha approaches 90,
     Actual_Heading approaches 180
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Borenstein, Johann, Everett, H.R., and Feng, Liqiang, *Navigating Mobile Robots: Systems and Techniques*, A. K. Peters, 1996.

Cox, Ingemar, and Leonard, John, "Dynamic Map Building for an Autonomous Mobile Robot", *Autonomous Mobile Robots: Perception, Mapping, and Navigation*, pp. 331-338, IEEE Computer Society Press, Los Alamitos, CA, 1991.

Crowley, J.L., "Dynamic World Modeling for an Intelligent Mobile Robot Using a Rotation Ultrasonic Ranging Device", *Proceeding IEEE International Conference on Robotics and Automation*, St. Louis, Missouri, pp. 128-135, 1985.

Crowley, J.L., "World Modeling and Position Estimation for a Mobile Robot Using Ultrasonic Ranging", *Proc. IEEE International Conference on Robots and Automation*, Scottsdale, AZ, pp. 674-680. 1989.

Drumheller, M., "Mobile Robot Localization Using Sonar", *IEEE Transactions on Pattern analysis and Machine Intelligence*, vol. PAMI-9, no. 2, pp. 325-332, 1987.

Elfes, Alberto, and Matthies, Larry, "Integration of Sonar and Stereo Range Data Using a Grid-Based Representation", *Autonomous Mobile Robots: Perception, Mapping, and Navigation*, pp. 234, IEEE Computer Society Press, Los Alamitos, CA, 1991.

Elfes, Alberto, "Sonar-Based Real-World Mapping and Navigation", *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 3. pp. 149-165, 1987.

Everett, H.R. and others, *Modeling the Environment of a Mobile Security Robot*, Naval Ocean Systems Center, San Diego, CA, 1990.

Everett, H.R. and others, *Survey of Collision Avoidance and Ranging Sensors for Mobile Robots*, Naval Ocean Systems Center, San Diego, CA, 1992.

Jarvis, R.A., "A Perspective on Range Finding Techniques for Computer Vision", *Autonomous Mobile Robots: Perception, Mapping, and Navigation*, pp. 20, IEEE Computer Society Press, Los Alamitos, CA, 1991.

Lochner, Jane, *Analysis and improvement of an Ultrasonic Sonar System on an Autonomous Mobile Robot*, Thesis, Naval Postgraduate School, 1994.

MacPherson, David, *Automated Cartography by an Autonomous Mobile Robot using Ultrasonic Range Finders*, Dissertation, Naval Postgraduate School, 1993.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 2060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   4111 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr. Yutaka Kanayama, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Professor Thomas Wu, Code CS/WQ 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Mark Merrell 1
   12385 Tebo Avenue
   Chino, CA 91710